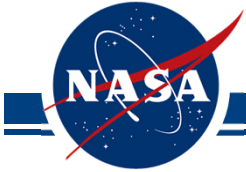


The Design of a Fault-Tolerant, Real-Time, Multi-Core Computer System

Kim P. Gostelow

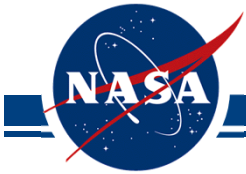
Jet Propulsion Laboratory, California Institute of Technology

March 9, 2011



The Vision

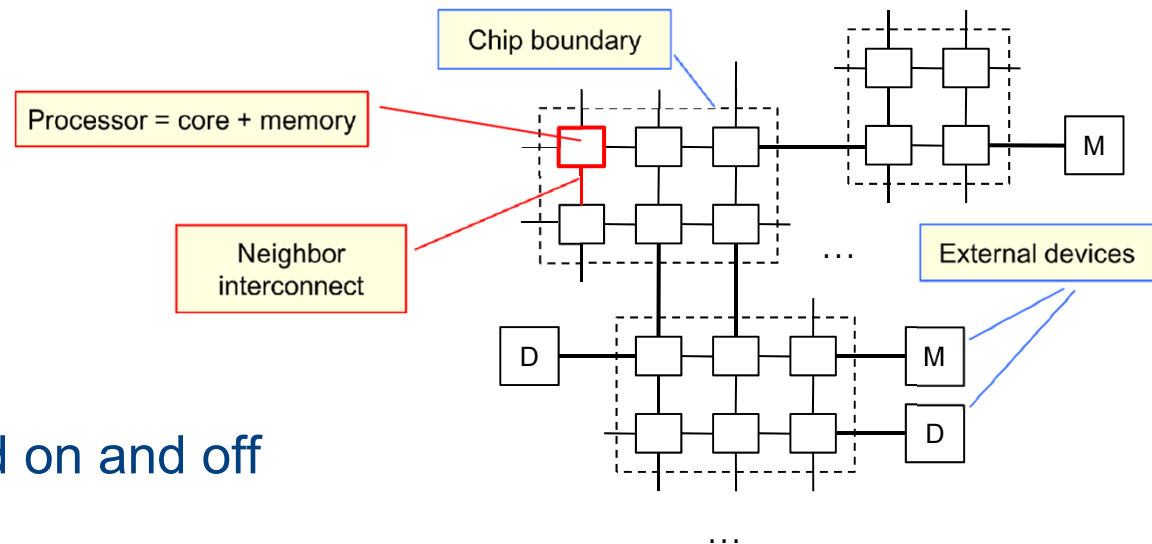
- Look to when there are thousands of cores on a spacecraft
 - Expectation
 - Faulty core=> computations move to another core
 - Reduce power => performance slows, but does not quit
 - Computations reorganize in real-time
 - Introspective
 - Little or no consideration needed by the programmer

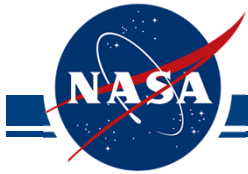


Machine Architecture

- Large number of cores per chip
 - No shared memory visible to the programmer
 - Any shared memory is for internal purposes (e.g., message passing)
- Cores communicate with neighbors via high-speed, message-passing links

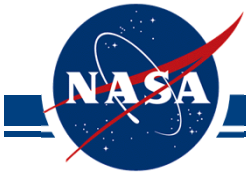
- Cores and links
 - May fail
 - May be powered on and off





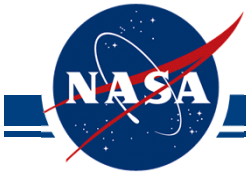
The Issue

- The above can be achieved now, but only by costly, special-case programming
- Programmers should not spend their time orchestrating intricate (and brittle) data arrangements and code
 - It breaks when processors fail
 - It should not be part of the job
- We want the machine, without intervention, without programmer's special attention, to re-organize its work automatically in the face of cores and links failing/re-appearing at random, in real-time.



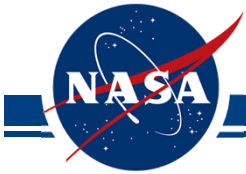
A Solution: (Mostly) Functional Programming

Von Neumann (Clocked sequential circuit)	Functional (Asynchronous circuit)
An instruction executes when the program counter reaches it.	The function executes when the required data arrives.
Instructions manipulate the contents of memory cells.	Variables are mathematical variables, not memory cells (contents cannot change once computed).
	No side-effects, no shared memory, no semaphores.



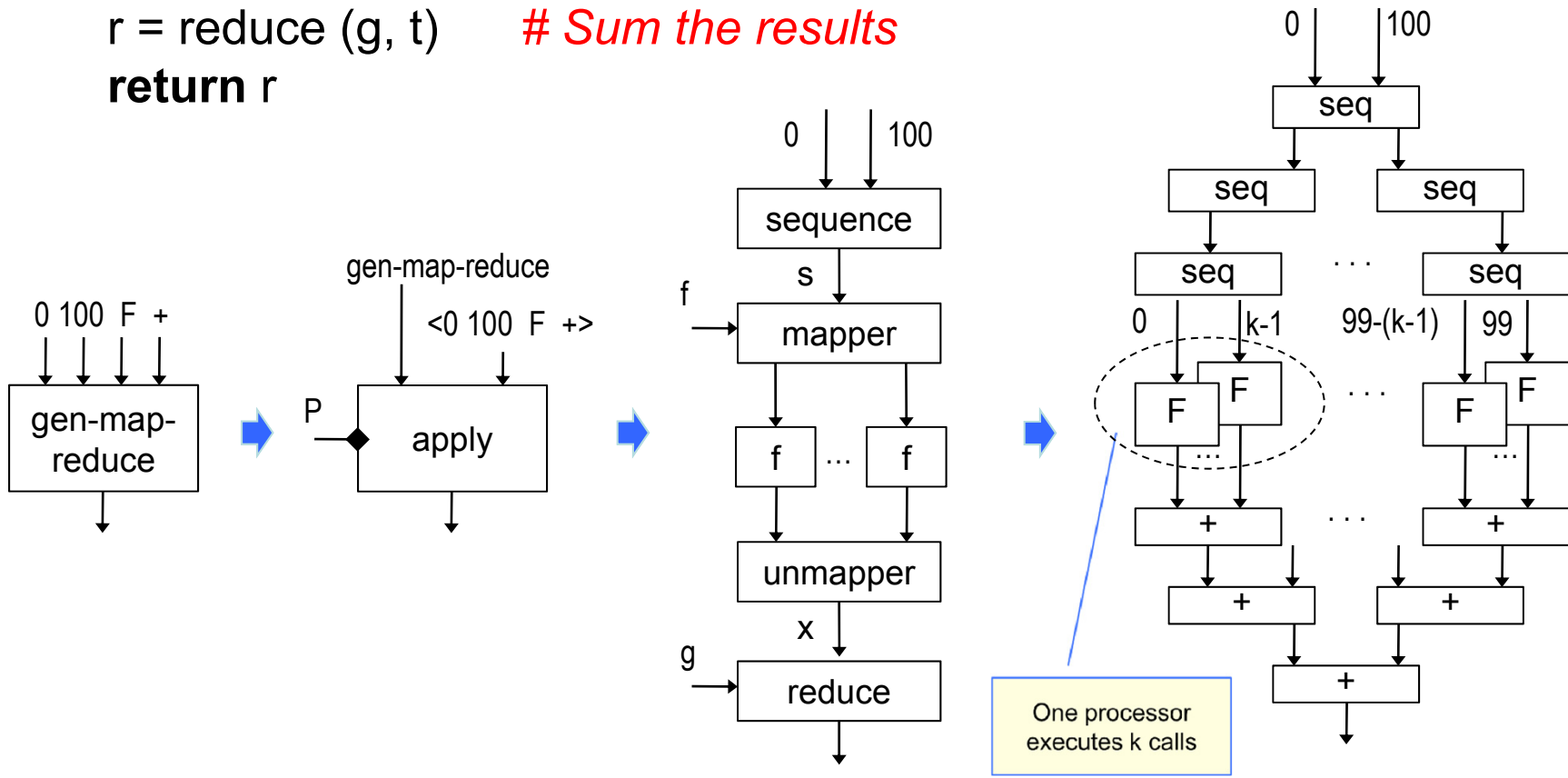
Properties of Functional Programs

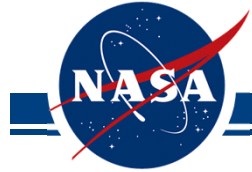
- Simple parallelism: Any two non-overlapping expressions can be executed in parallel
 - Consider: $f(x) + f(y)$
 - von Neumann: f may have memory and the result may depend upon the order of execution
 - Functional: order of execution is irrelevant; there is no shared memory; you always get the same answer for the same arguments
- Simple analyses:
 - Always get the same outputs given the same inputs
 - Can copy, stop, move, restart, ... without concern
 - Can re-execute any function at any time
 - Can throw computations away and re-execute
 - Can move computations without regard to memory
 - For example, can execute any function in TMR at any time



Example: generate-map-reduce

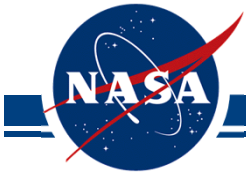
```
define gen-map-reduce (a, b, f, g) =  
  s = seq (a, b)      # Generate the integers from a up to b  
  t = map (f, s)       # Produce f(i) for each i  
  r = reduce (g, t)    # Sum the results  
return r
```





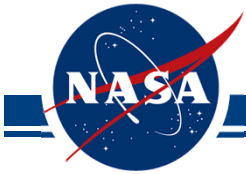
Internals

- Each box is an *Actor* [Hewitt]
 - Function application creates an *actor frame* for each actor in the function.
 - Each actor frame is sent to its assigned processor for execution.
- Actor execution
 - Actor arguments arrive at the processor and are kept in the actor frame.
 - When all arguments have arrived, the actor executes.
 - The actor sends its results to the next actor.
 - The actor disappears.



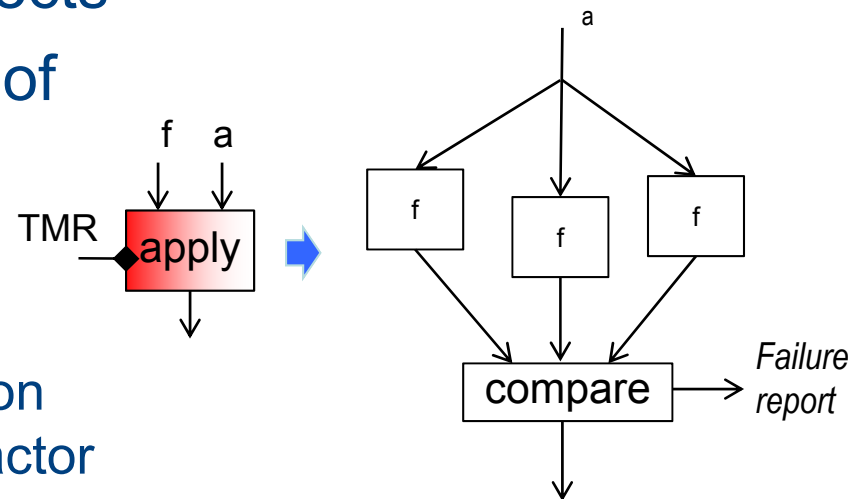
Moving Computations

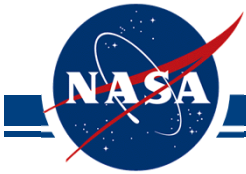
- The extra argument to *apply* gives the processors to use for that call
 - Higher-level calls pass allocations to lower-level calls
 - Top-level allocation is from the system supervisor
 - Responsive to faults and power availability
- A new processor allocation/assignment can occur at each call
- For long-running functions, the values can be updated and inner function calls can respond with new assignments
 - An update procedure moves in-process actor frames



Fault Tolerance

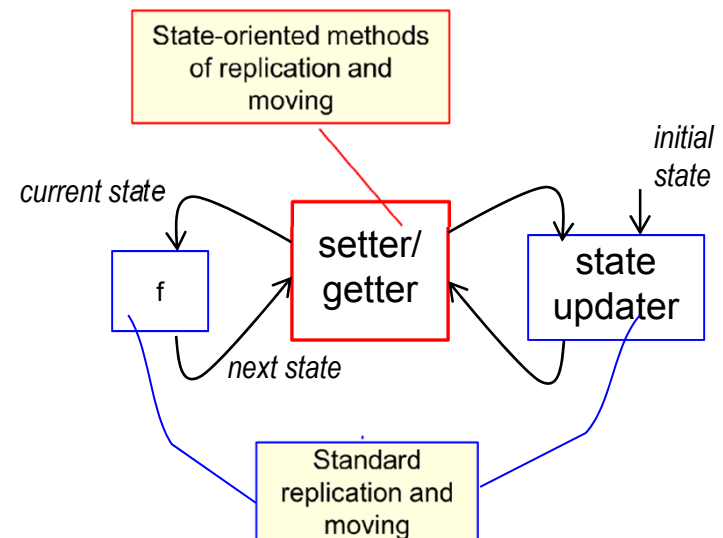
- Functions have no side-effects
- TMR: *Apply* runs N copies of the function instead of just one:
 - Triplicate the actor frame
 - Change the actor's destination actor to a TMR comparator actor
 - The comparator checks the results
 - Identical: send the result on to the original result actor.
 - Different: carry out fault recovery

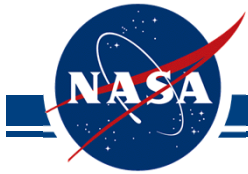




Non-functional Code

- State is treated differently (not theoretically, but as a practical matter)
- A small part of code
 - Recognized in the source language so no analysis needed
- A function applied to a state value can be replicated, moved, restarted, ...





Summary

- Very large number of cores are coming to spacecraft
 - Computations need to adjust automatically to power and hardware failure
 - With little to no programmer assistance
- Thesis: (mostly) functional programming an approach
 - Recognizing the special role played by state.
- Current work: building a simulator to find and measure effective methods of adjusting computations to power and fault circumstances.